Rozhledy matematicko-fyzikální

Zdeněk Dvořák; Martin Mareš; David Matoušek

Recepty z programátorské kuchařky Korespondenčního semináře z programování, VIII. část

Rozhledy matematicko-fyzikální, Vol. 84 (2009), No. 1, 26–34

Persistent URL: http://dml.cz/dmlcz/146090

Recepty z programátorské kuchařky Korespondenčního semináře z programování, VIII. část*)

Zdeněk Dvořák, Martin Mareš, David Matoušek, MFF UK Praha

Abstract. We present several well-known applications of Divide and Conquer Method (dividing the problem to smaller subproblems and composing their solutions to a solution of the original problem): a linear-time algorithm for finding the median of a sequence, and a subquadratic algorithm for multiplication of long numbers.

V tomto dílu programátorské kuchařky se budeme zabývat algoritmy založenými na metodě *Rozděl a panuj*. A tak by se slušelo začít tím, jaká je myšlenka této metody: Často se setkáme s úlohami, které lze snadno rozdělit na nějaké menší úlohy a z jejich výsledků zase složit výsledek původní velké úlohy. Přitom menší úlohy můžeme řešit opět stejným algoritmem (zavoláme ho rekurzivně), jestliže nejsou dostatečně malé, abychom jejich výsledek nalezli triviálně bez jakéhokoliv počítání. Zkrátka jak říkali staří římští císaři: Divide et impera. Uveďme si pro začátek jeden staronový příklad.

1. QuickSort

QuickSort je algoritmus pro třídění posloupnosti prvků. Už o něm byla jednou řeč ve druhém dílu kuchařky [4]. Tentokrát se na něj podíváme trochu podrobněji a navíc nám poslouží jako ingredience pro další algoritmy.

QuickSort v každém svém kroku zvolí nějaký prvek (budeme mu říkat *pivot*) a přerovná prvky v posloupnosti tak, aby napravo od pivota byly pouze prvky větší než pivot a nalevo pouze menší. Pokud se vyskytnou prvky rovné pivotu, můžeme si dle libosti vybrat jak levou, tak pravou stranu posloupnosti, funkčnost algoritmu to nijak neovlivní.

*) Informace o Korespondečním semináři z programování pořádaného Matematicko-fyzikální fakultou Univerzity Karlovy v Praze lze nalézt na webové stránce `http://ksp.mff.cuni.cz`.

Tento postup pak rekurzivně zopakujeme zvlášť pro prvky nalevo a zvlášť pro prvky napravo od pivota, a tak získáme setříděnou posloupnost.

Implementací QuickSortu je mnoho a mimo jiné se liší způsobem volby pivota. My si předvedeme jinou, než jsme ukazovali ve druhém dílu kuchařky (hlavně proto, že se nám z ní budou snadno odvozovat další algoritmy), a pro jednoduchost budeme jako pivota volit poslední prvek zkoumaného úseku:

```
{budeme třídit takováto pole}
type Pole=array[1..MaxN] of Integer;
{přerovnávací procedura pro úsek a[l..r]}
function prer(a:Pole; l,r:Integer):Integer;
var i,j,x,q:Integer;
begin
  {pivotem se stane poslední prvek úseku}
  x:=a[r];   {hodnota pivota}
  i:=l-1;   {a[i] bude vždy poslední <= pivotovi}
  {samotné přerovnávání}
  for j:=l to r-1 do
    if a[j]<=x then   {právě probíraný prvek}
    begin   {menší/rovný hodnotě pivota}
    Inc(i);   {pak zvyš ukazatel}
    {a proveď přerovnání prvku}
    q:=a[j]; a[j]:=a[i]; a[i]:=q;
  end;
  {nakonec přesuneme pivota za poslední <=}
  q:=a[r]; a[r]:=a[i+1]; a[i+1]:=q;
  prer:=i+1;   {vrátíme novou pozici pivota}
end;
{hlavní třídící procedura, třídí a[l..r]}
procedure QuickSort(a:Pole; l,r:Integer);
var m:Integer;
begin
  if l<r then   {máme ještě co dělat?}
  begin
    m:=prer(l,r);   {přerovnej, m pozice pivota}
    QuickSort(l,m-1);   {setřiď prvky napravo}
    QuickSort(m+1,r);   {setřiď prvky nalevo}
  end;
end;
```

Bohužel volit pivota právě takto je docela nešikovné, protože se nám snadno může stát, že si vybereme nejmenší nebo největší prvek v úseku (rozmyslete si, jak by vypadala posloupnost, ve které to nastane pokaždé), takže dostaneme-li posloupnost délky N, rozdělíme ji na úseky délek $N-1$ a 1, načež pokračujeme s úsekem délky $N-1$, ten rozdělíme

na $N-2$ a 1 atd. Přitom pokaždé na přerovnání spotřebujeme čas lineární s velikostí úseku, celkem tedy $O(N+(N-1)+(N-2)+\ldots+1) = O(N^2)$.

Na druhou stranu pokud bychom si za pivota vybrali vždy *medián* z právě probíraných prvků (tj. prvek, který by se v setříděné posloupnosti nacházel uprostřed; pro sudý počet prvků zvolíme libovolný z obou prostředních prvků), dosáhneme daleko lepší složitosti $O(N \log N)$:

Přerovnávací část algoritmu běží v čase lineárním vůči počtu prvků, které máme přerovnat. V prvním kroku QuickSortu pracujeme s celou posloupností, čili přerovnáme celkem N prvků. Následuje rekurzivní volání pro levou a pravou část posloupnosti (obě mají délku nanejvýš $N/2$); přerovnávání v obou částech dohromady trvá opět $O(N)$ a vzniknou tím části dlouhé nejvýše $N/4$. Zanoříme-li se v rekurzi do hloubky k, pracujeme s částmi dlouhými nejvýše $N/2^k$ a součet jejich délek je nejvýše N (všechny části dohromady dají prvky vstupní posloupnosti bez těch, které jsme si už zvolili jako pivoty). V hloubce $\lceil \log_2 N \rceil$ už jsou všechny části nejvýše jednoprvkové, takže se rekurze zastaví. Celkem tedy máme $\lceil \log_2 N \rceil$ hladin (hloubek) a na každé z nich trávíme lineární čas, dohromady $O(N \log N)$.

V tomto důkazu jsme se ale dopustili jednoho podvodu: Zapomněli jsme na to, že také musíme medián umět najít. Jak z této nepříjemné situace ven?

- *Naučit se počítat medián.* Ale jak?
- *Spokojit se se „lžimediánem“.* Kdybychom si místo mediánu vybrali libovolný prvek, který bude v setříděné posloupnosti v prostřední polovině (čili alespoň čtvrtina prvků bude větší a alespoň čtvrtina menší než on), získáme také složitost $O(N \log N)$: Úsek délky N rozložíme na úseky, které budou mít délky nejvýše $3/4 \cdot N$, takže na k-té hladině budou úseky délek $\leq (3/4)^k \cdot N$. Hladin proto bude maximálně $\log_{4/3} N = O(\log N)$. Místo $1/4$ by fungovala i libovolná jiná konstanta, ale ani to nám nepomůže k tomu, abychom uměli lžimedián najít.
- *Recyklovat pravidlo* typu „vezmi poslední prvek“ a jen ho trochu vylepšit (například si pivot zvolit jako medián z prvního, prostředního a posledního prvku posloupnosti). To bohužel nebude fungovat, protože pokud budeme při výběru pivota testovat jen konstantní počet prvků, budou existovat vstupy, pro které toto pravidlo bude dávat kvadratickou složitost. Obvykle však lze dokázat, že takových vstupů je „málo“, proto se toto řešení prakticky často používá.

- *Volit pivota náhodně* ze všech prvků zkoumaného úseku. K náhodné volbě samozřejmě potřebujeme náhodný generátor a s těmi je to svízelné, ale zkusme na chvíli věřit, že jeden takový máme nebo alespoň že máme něco s podobnými vlastnostmi. Jak nám pomůže? Náhodně zvolený pivot nebude sice přesně uprostřed, ale s pravděpodobností 1/2 to bude lžimedián, takže po průměrně dvou hladinách se ke lžimediánu dopracujeme. Proto časová složitost takovéhoto randomizovaného QuickSortu bude v průměru nanejvýš dvakrát větší, než lžimediánového QuickSortu, čili také $O(N \log N)$. Zatímco fixní pravidlo nám dalo dobrý čas pro průměrný vstup, ale existovaly vstupy, na kterých bylo pomalé, randomizování nám dává dobrý průměrný čas pro všechny možné vstupy.

2. Hledání k-tého nejmenšího prvku

Nad QuickSortem jsme zvítězili, ale současně jsme při tom zjistili, že neumíme rychle najít medián. Zkusíme nyní vyřešit obecnější problém: Najít k-tý nejmenší prvek (medián dostáváme pro $k = \lfloor N/2 \rfloor$).

První řešení této úlohy se nabízí samo. Načteme posloupnost do pole, prvky pole setřídíme nějakým rychlým algoritmem a kýžený k-tý nejmenší prvek nalezneme na k-té pozici v nyní již setříděném poli. Nicméně, jestliže prvky, které máme na vstupu, umíme pouze porovnat, pak nedosáhneme lepší časové složitosti (a to ani v průměrném případě) než $O(N \log N)$ – rychleji třídit nelze, důkaz můžete najít opět ve druhé kuchařce.

O něco rychlejší řešení (Hoare [2]) je založeno na výše zmíněném algoritmu QuickSort (často se mu proto říká QuickSelect). Opět si vybereme pivota a posloupnost rozdělíme na prvky menší než pivot, pivota a prvky větší než pivot (pro jednoduchost budeme předpokládat, že žádné dva prvky posloupnosti nejsou stejné). Jestliže je právě $k - 1$ prvků menších než pivot, pak pivot je hledaný k-tý nejmenší prvek posloupnosti. Zbývají dva případy, kdy tomu tak není. Jestliže je pozice pivota v posloupnosti větší než k, pak rekurzivně nalezneme k-tý nejmenší prvek mezi prvky menšími než pivot. V opačném případě, kdy je pozice pivota menší než k, je hledaný prvek větší než pivot. Mezi těmito prvky však nebudeme hledat k-tý nejmenší prvek, ale $(k-p)$-tý nejmenší prvek, kde p je pozice pivota v posloupnosti (počet prvků menších než pivot $+1$).

Časovou složitost rozebereme podobně jako u QuickSortu. Nešikovná volba pivota dává opět v nejhorším případě kvadratickou složitost. Po-

kud bychom naopak volili za pivota medián, budeme nejprve přerovnávat N prvků, pak jich zbude nejvýše $N/2$, pak nejvýše $N/4$ atd., což dohromady dává složitost $\mathrm{O}(N + N/2 + N/4 + \ldots + 1) = \mathrm{O}(N)$. Pro lžimedián dostaneme rovněž lineární složitost a stejně jako u QuickSortu můžeme nahlédnout, že náhodnou volbou pivota dostaneme v průměru stejnou časovou složitost jako se lžimediánem.

S použitím přerovnávací procedury QuickSortu je program velmi jednoduchý:

```
function kty(var a:Pole; l,r,k:Integer):Integer;
var x,z:Integer;
begin
  x:=prer(a,l,r);   {přerovnej, x je pozice pivota}
  z:=x-l+1;   {pozice pivota vzhledem k [l..r]}
  if k=z then kty:=a[x]   {k-tý nejmenší je pivot}
  else if k<z then
    kty:=kty(a,l,x-1,k)   {k-tý nejmenší je nalevo}
  else kty:=kty(a,x+1,r,k-z);   {napravo}
end;
```

3. k-tý nejmenší podruhé, tentokrát lineárně a bez náhody

Existuje i algoritmus (Blum at al. [1]), který řeší naši úlohu lineárně, a to i v nejhorším případě. Je založený na ďábelském triku: Zvolit vhodného pivota (jak ukážeme, bude to jeden ze lžimediánů) rekurzivním voláním téhož algoritmu. Podrobněji:

1. Pokud jsme dostali méně než 6 prvků, použijeme nějaký triviální algoritmus, například si posloupnost setřídíme a vrátíme k-tý prvek setříděné posloupnosti.
2. Jinak rozdělíme prvky posloupnosti na pětice; pokud není počet prvků dělitelný pěti, poslední pětici necháme nekompletní.
3. Spočítáme medián každé pětice (v konstantním čase), například opět jejich setříděním.
4. Máme tedy nanejvýš $N/5$ mediánů pětic. V nich rekurzivně najdeme medián m (označíme mediány pětic za novou posloupnost a začneme pro ni opět od prvního bodu).
5. Použijeme m jako pivota a rozdělíme vstupní posloupnost na prvky menší než m a prvky větší než m. Nechť je z počet prvků s menší hodnotou, než má pivot.
6. Stejně jako u předchozího algoritmu, jestliže $k = z+1$, pak pivot m je k-tým nejmenším prvkem posloupnosti. V případě, že tomu tak není a $k < z+1$, budeme hledat k-tý nejmenší prvek mezi prvními z členy

posloupnosti, které jsou menší než m, a jestliže $k > z + 1$, budeme hledat $(k - z - 1)$-ní nejmenší prvek mezi prvky většími než m.

V Pascalu:

```
{potřebujeme přerovnávací funkci, která}
{dostane hodnotu pivota jako parametr}
function prerp(var a:Pole; l,r,m:Integer):Integer;
var q,p:Integer;
begin
  {nalezneme pozici pivota}
  p:=l;
  while a[p]<>m do inc(p);
  {pivota prohodíme s posledním prvkem}
  q:=a[p]; a[p]:=a[r]; a[r]:=q;
  {a zavoláme původní přerovnávací fci}
  prerp := prer(a,l,r);
end;
{hledání k-tého nejmenšího prvku z a[l..r]}
function kth(var a:Pole; l,r,k:Integer):Integer;
var medp:Pole;   {pole pro mediány pětic}
  i,j,q,x,pocet,m,z:Integer;
begin
  pocet:=r-l+1;   {s kolika prvky pracujeme}
  if pocet<=1 then   {pouze jeden prvek?}
    kth:=a[l]   {výsledek nemůže být jiný}
  else if pocet<6 then begin   {méně než 6 prvků}
    QuickSort(a,l,r);
    kth:=a[l+k-1];
    end
  else begin {mnoho prvků, jde to tuhého, rozdělíme je do pětic}
    q:=1;   {zatím máme jednu pětici}
    i:=l;   {levý okraj první pětice}
    j:=i+4;   {pravý okraj první pětice}
    while j<=r do begin   {procházíme celé pětice}
       QuickSort(a,i,j);
       medp[q]:=a[i+2];   {medián pětice}
       Inc(q);   {zvyš počet pětic}
       Inc(i,5);   {nastav levý okraj pětice}
       Inc(j,5);   {nastav pravý okraj pětice}
    end;
  {případnou neúplnou pětici můžeme ignorovat}
  {najdeme medián mediánů pětic}
  m:=kth(medp,1,q-1,q div 2);
  {přerovnej a zjisti, kde skončil pivot}
  x:=prer(a,l,r,m);
  z:=x-l+1;   {pozice vzhledem k [l..r]}
```

```
    if k=z then kth:=m   {k-tý nejmenší je pivot}
    else if k<z then kth:=kth(a,l,x-1,k)   {k-tý nejmenší nalevo}
    else kth:=kth(a,x+1,r,k-z);   {napravo}
    end;
end;
```

Zbývá dokázat, že tato dvojitá rekurze má lineární složitost. Zkusme se proto podívat, kolik prvků posloupnosti po přerovnání je větších než prvek m. Všech pětic je $N/5$ a alespoň polovina z nich (tedy $N/10$) má medián menší než m. V každé takové pětici pak navíc najdeme dva prvky menší než medián pětice, takže celkem existuje alespoň $3/10 \cdot N$ prvků menších než m. Větších tedy může být maximálně $7/10 \cdot N$. Symetricky ukážeme, že i menších prvků může být nejvýše $7/10 \cdot N$.

Rozdělení na pětice, hledání mediánů pětic a přerovnávání trvá lineárně, tedy nejvýše cN kroků pro nějakou konstantu $c > 0$. Navíc algoritmus dvakrát rekurzivně volá sám sebe: Nejprve pro $N/5$ mediánů pětic, pak pro nanejvýš $7/10 \cdot N$ prvků před/za pivotem. Pro celkovou časovou složitost $t(N)$ našeho algoritmu tedy platí

$$t(N) \leq cN + t(N/5) + t(7/10 \cdot N).$$

Nyní zbývá tuto rekurzivní nerovnici vyřešit, což provedeme drobným úskokem: uhodneme, že výsledkem bude lineární funkce, tedy že $t(N) = dN$ pro nějaké $d > 0$. Dostaneme

$$dN \leq (c + 1/5 \cdot d + 7/10 \cdot d) \cdot N.$$

To platí například pro $d = 10c$, takže $t(N) = \mathrm{O}(N)$.

4. Násobení dlouhých čísel

Dalším pěkným příkladem na metodu rozděl a panuj je násobení dlouhých čísel – tak dlouhých, že se už nevejdou do integeru, takže s nimi musíme počítat po číslicích (ať už v jakékoliv soustavě – my volíme desítkovou, často se hodí třeba 256-ková). Klasickým „školním“ algoritmem pro násobení na papíře to zvládneme na kvadratický počet operací, zde si předvedeme efektivnější způsob (Karatsuba a Ofman [3]).

Libovolné $2N$-ciferné číslo můžeme zapsat jako $10^N A + B$, kde A a B jsou N-ciferná. Součin dvou takových čísel pak bude:

$$(10^N A + B) \cdot (10^N C + D) = 10^{2N} AC + 10^N (AD + BC) + BD$$

Sčítat dokážeme v lineárním čase, násobit mocninou deseti také (dopíšeme příslušný počet nul na konec čísla), N-ciferná čísla budeme násobit rekurzivním zavoláním téhož algoritmu. Pro časovou složitost tedy bude platit $t(N) = cN + 4t(N/2)$. Nyní tuto rovnici můžeme snadno vyřešit, ale ani to dělat nebudeme, neboť nám vyjde, že $t(N) \approx N^2$, čili jsme si oproti původnímu algoritmu vůbec nepomohli.

Nicméně, místo čtyř násobení čísel poloviční délky nám stačí pouze tři: Spočteme AC, BD a $(A+B)\cdot(C+D) = AC+AD+BC+BD$, přičemž pokud od posledního součinu odečteme AC a BD, dostaneme $AD+BC$, které jsme předtím počítali dvěma násobeními. Časová složitost nyní bude $t(N) = c'N + 3t(N/2)$. Konstanta c' je o něco větší než c, protože přibylo sčítání a odčítání. Zvolme si pro jednoduchost jednotku času tak, aby bylo $c' = 1$.

Jak tuto rovnici vyřešíme? Zkusíme ji dosadit do sebe samé a pozorovat, co se bude dít:

$$\begin{aligned} t(N) &= N + 3(N/2 + 3t(N/4)) = N + 3/2 \cdot N + 9t(N/4) = \\ &= N + 3/2 \cdot N + 9/4 \cdot N + 27t(N/8) = \\ &= N + 3/2 \cdot N + \ldots + 3^{k-1}/2^{k-1} \cdot N + 3^k t(N/2^k) \end{aligned}$$

Pokud zvolíme $k = \log_2 N$, vyjde $N/2^k = 1$, čili $t(N/2^k) = t(1) = d$, kde d je nějaká konstanta. To znamená, že

$$t(N) = N \cdot (1 + 3/2 + 9/4 + \ldots + (3/2)^{k-1}) + 3^k d.$$

Výraz v závorce je součet prvních k členů geometrické řady s kvocientem $3/2$, čili:

$$\frac{(3/2)^k - 1}{(3/2 - 1)} = \mathrm{O}((3/2)^k)$$

Tato funkce však roste pomaleji než zbylý člen $3^k d$, takže ji můžeme zanedbat a zabývat se pouze oním posledním členem

$$3^k = 2^{k \log_2 3} = 2^{\log_2 n \cdot \log_2 3} = (2^{\log_2 n})^{\log_2 3} = n^{\log_2 3} \approx n^{1.58}.$$

Konstanta d se nám „schová do O-čka“, takže algoritmus má časovou složitost přibližně $\mathrm{O}(n^{1.58})$. Existují i rychlejší algoritmy (např. Schönhage–Strassenův algoritmus [6] s časovou složitostí $\mathrm{O}(n \log n \log\log n)$), ale jsou mnohem složitější a pro malá n se nevyplatí – zatímco popsaný algoritmus je rychlejší než jednoduchý kvadratický algoritmus

pro $n \approx 100$, Schönhage–Strassenův algoritmus je rychlejší až pro $n \approx 20\,000$.

5. Rozmyslete si

- Při hledání k-tého nejmenšího prvku jsme předpokládali, že všechny prvky jsou různé. Prohlédněte si algoritmy pozorně a rozmyslete si, že budou fungovat i bez toho. Opravdu?
- Proč jsme zvolili zrovna pětice? Jak by to dopadlo pro trojice? A jak pro sedmice? Fungoval by takový algoritmus? Byl by také lineární?
- Ve výpočtu $t(N)$ jsme si nedali pozor na neúplné pětice a také jsme předpokládali, že pětic je sudý počet. Ono se totiž nic zlého nemůže stát. Jak se to snadno nahlédne? Proč nestačí na začátku doplnit vstup „nekonečny“ na délku, která je mocninou deseti?
- Kdybychom neuhodli, že $t(N)$ je lineární, jak by se na to dalo přijít?
- Ještě jednou QuickSort: Představte si, že budujete binární vyhledávací strom (viz například čtvrtou kuchařku [5]) vkládáním prvků v náhodném pořadí. Obecně nemusí být vyvážený, ale v průměru v něm půjde vyhledávat v čase $\mathrm{O}(\log N)$. Žádný div: Stromy, které nám vzniknou, odpovídají přesně možným průběhům QuickSortu.

Literatura

[1] Blum, M., Floyd, R. W., Pratt, V., Rivest, R., Tarjan, R.: Time bounds for selection. *J. Comput. System Sci.* **7** (1973), s. 448–461.

[2] Hoare, C. A. R.: Partition: Algorithm 63, Quicksort: Algorithm 64, Find: Algorithm 65. *Comm. ACM* **4**, 7 (1961), s. 321–322.

[3] Karatsuba, A., Ofman, Yu.: Multiplication of Many-Digital Numbers by Automatic Computers. *Doklady Akad. Nauk SSSR* **145** (1962), s. 293–294. Translation in *Physics–Doklady* **7** (1963), 595–596.

[4] Kráľ, D., Mareš, M., Valla, T.: Recepty z programátorské kuchařky Korespondenčního semináře z programování – II. část. *Rozhledy matematicko-fyzikální* **80**, 2 (2005), s. 25–35.

[5] Kráľ, D., Mareš, M., Straka, M.: Recepty z programátorské kuchařky Korespondenčního semináře z programování – IV. část. *Rozhledy matematicko-fyzikální* **82**, 1 (2007), s. 22–35.

[6] Schönhage, A., Strassen, V.: Schnelle Multiplikation großer Zahlen. *Computing* **7** (1971), s. 281–292.